山东大学
SHANDONG UNIVERSITY

崇新学堂

2024 — 2025 学年第一学期

实验报告

课程名称：Introduction to EECS Lab
实验名称：DL11 - Robots in Hallways
学生姓名：胡君安、陈焕斌、黄颢
实验时间：2024 年 12 月 12日

DL11 - Robots in Hallways

1. Introduction

When dealing with systems with unobservable internal states, state estimation helps us infer these hidden states based on observable data. Examples include:

- A copy machine: Hidden state is the machinery condition; actions are making copies; observations are copy quality.
- A robot in a hallway: Hidden state is the robot's location; actions are moving east or west; observations are wall colors.
- A video game player: Hidden state is the target monster; observations are the player's moves.

State estimation uses inputs (actions) and observations to compute a probability distribution over hidden states. We'll use this to estimate a robot's pose with noisy sonar and odometry readings.

Key components for probabilistic state estimation:

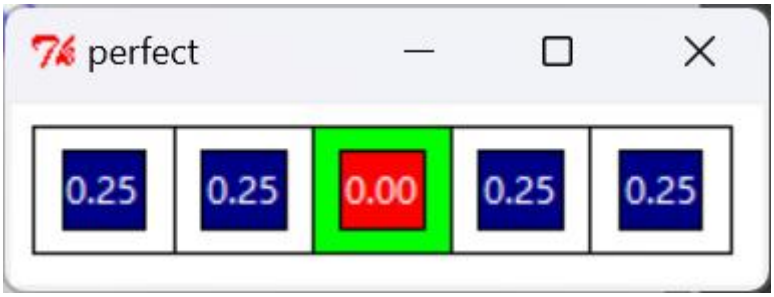
1. **Initial distribution**: Probability of each state being the initial one.
2. **Observation model**: Probability of observations given by the state.
3. **Transition model**: Probability of moving to each state at time (t+1) given the state and action at time (t).

2. Experimental step

Step 1: Move the Robot(Perfect)

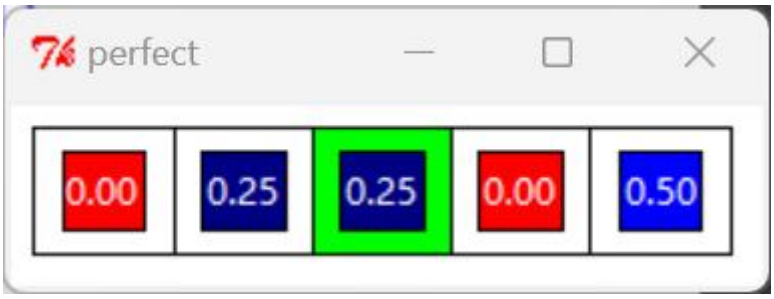
> #### Check Yourself 1
>
> Move the robot around in the perfect simulator. Be sure you understand what the colors representing the belief state mean and that the numbers being printed out in the Python shell make sense. Feel free to ask a staff member for clarification.

```
>>> p.run(10)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 0
after obs white DDist(0: 0.250000, 1: 0.250000, 3: 0.250000, 4: 0.250000)
after trans 0 DDist(0: 0.250000, 1: 0.250000, 3: 0.250000, 4: 0.250000)
```



When not moving, the generated distribution is up. That is, when detecting that the robot itself is white, the probability distribution model is displayed based on the possible position of the robot after moving 0 bits (i.e. not moving). It can be seen because the detected self-room is white, it is impossible to be in the green room, and the probability distribution of other white rooms is equal.

```
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(0: 0.250000, 1: 0.250000, 3: 0.250000, 4: 0.250000)
after trans 1 DDist(1: 0.250000, 2: 0.250000, 4: 0.500000)
```



Move one bit to the right, and the generated distribution is up. That is, under the premise of detecting itself as white, move one bit to the right and display the probability distribution model according to the possible position of the robot. It can be seen that because it moves one bit to the right, the probability of being in the leftmost room is 0; because it is detected as white and moves one bit to the right, that is, it is initially impossible to be in the green room, then moving one bit to the right cannot be in the third white room, and the probability is 0.

```
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): -1
after obs white DDist(0: 0.250000, 1: 0.250000, 3: 0.250000, 4: 0.250000)
after trans -1 DDist(0: 0.500000, 2: 0.250000, 3: 0.250000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): -1
after obs white DDist(0: 0.666667, 3: 0.333333)
after trans -1 DDist(0: 0.666667, 2: 0.333333)
```

Move left twice in a row. Similarly, we explain its final probability distribution: because it is detected as white on the second move, the probability of the second white room is 0; because it has moved left twice, the probability of the third and fourth white rooms is 0.

Step 2:Move the Robot(Noisy)

Check Yourself 2

Move the robot around in the noisy simulator. Be sure you understand what the colors mean, and have a basic idea of what might be going on. Feel free to ask a staff member for clarification.

```
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 0
after obs white DDist(0: 0.248804, 1: 0.248804, 2: 0.004785, 3: 0.248804, 4: 0.2
48804)
after trans 0 DDist(0: 0.248804, 1: 0.224402, 2: 0.053589, 3: 0.224402, 4: 0.248
804)
```

Because it is a noise model, it is also possible to move right or left to the green room, and the probability of the green room is not 0 at this time. Its specific action mode is the same as before, but it will not accurately move its position according to the input movement.

Step 3: Observations for Each Room

Define an observation noise model where white and green are indistinguishable, meaning the robot is equally likely to see white or green when in a white or green square. All other colors are observed perfectly. This function should return a dist.DDist over observed colors, given the actual color as an argument.

```
Python
def whiteEqGreenObsDist(actualColor):
    d = {}
    for observedColor in ['white', 'green']:
        if observedColor == actualColor:
            d[actualColor] = 0.5
        else:
            d[observedColor] = 1.0
    return dist.DDist(d)
```

Give an observation noise distribution in which white always looks green, green always looks white, and all other colors are observed perfectly.

```
Python
def whiteVsGreenObsDist(actualColor):
    d = {}
    if actualColor == 'white':
        d['green'] = 1.0
    elif actualColor == 'green':
        d['white'] = 1.0
    else:
        d[actualColor] = 1.0
    return dist.DDist(d)
```

Define an observation noise distribution in which there is a probability of 0.8 of observing the actual color of a room, and there is a probability of 0.2 of seeing one of the remaining colors (equally likely which other color you see). All possible colors are available in the list called possibleColors.

```
Python
def noisyObs(actualColor):
    possibleColors = ['black', 'white', 'red', 'green',
'blue']
    #possibleColors = testHallway
    d = {}
    for observedColor in possibleColors:
        if observedColor == actualColor:
            d[observedColor] = 0.8
        else:
            d[observedColor] = 0.2 / (len(possibleColors) -
1)
    return dist.DDist(d)
```

The observation noise distributions defined so far encode the error model for observation independently of location. However, we need to obtain a probability distribution over possible observations (colors) at specific locations along the hallway. The makeObservationModel function allows us to construct such a model, given a list of colors (e.g., standardHallway) and an observation noise distribution (e.g., noisyObs). Enter the expression to create the full observation model for standardHallway using the noisyObs distribution.

```
Python
standardHallway = ['white', 'white', 'green', 'white', 'white']
noisyObsMode1 = makeObservationModel(standardHallway,noisyObs)
```

Check Yourself 3

Just to be sure you understand the observation models, consider a world with two rooms: room R0 is actually green and room R1 is actually white.

- With a perfect sensor, what is the probability distribution over observations for each room?

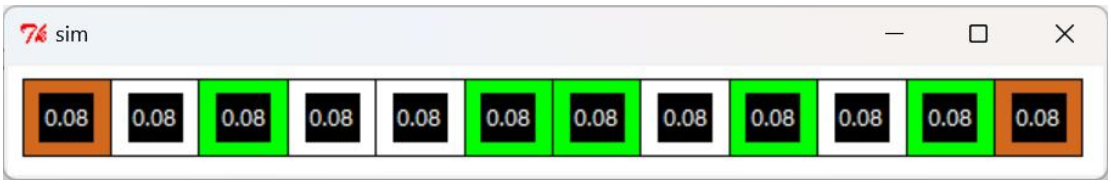
- With `whiteEqGreenObsDist`, what is the probability distribution over observations for each room?

- With `whiteVsGreenObsDist`, what is the probability distribution over observations for each room?

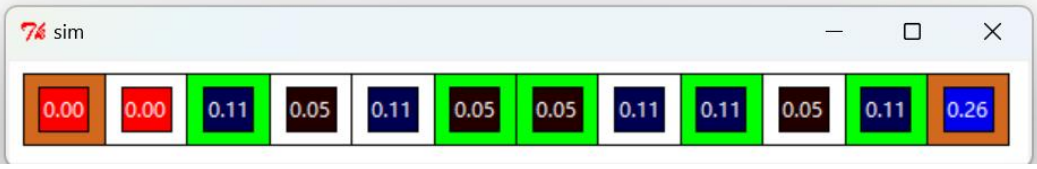
Step 4: Testhallway

We tested our whiteEqGreenObsDist and whiteVsGreenObsDist functions to observe their impact on the belief state. We pasted our definitions of these functions from the tutor into our designLab11Work.py file. We used the perfect motion models as shown below, ensuring we used something like whiteEqGreenObsDist as the third argument. The variables actions, standardDynamics, and perfectTransNoiseModel were already defined for us in designLab11Work.py.

```
>>> w = makeSim(testHallway, actions,
whiteEqGreenObsDist,
standardDynamics, perfectTransNoiseModel)
>>> w.run(50)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ):
```



```
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 2
after obs white DDist(0: 0.105263, 1: 0.052632, 2: 0.105263, 3: 0.052632, 4: 0.0
52632, 5: 0.105263, 6: 0.105263, 7: 0.052632, 8: 0.105263, 9: 0.052632, 10: 0.10
5263, 11: 0.105263)
after trans 2 DDist(2: 0.105263, 3: 0.052632, 4: 0.105263, 5: 0.052632, 6: 0.052
632, 7: 0.105263, 8: 0.105263, 9: 0.052632, 10: 0.105263, 11: 0.263158)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ):
```



Step 5: Transition Model for Each Room

相关代码如下:

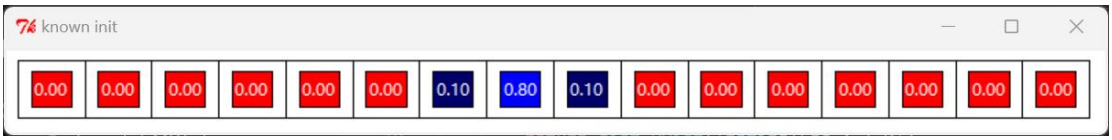
```
Python
def ringDynamics(loc, act, hallwayLength):
    return (loc + act) % hallwayLength

def leftSlipTrans(nominalLoc, hallwayLength):
    if nominalLoc != 0:
        return dist.DDist({nominalLoc : 0.9, nominalLoc - 1 :
0.1})
    else:
        return dist.DDist({nominalLoc : 1.0})

def noisyTrans(nominalLoc, hallwayLength):
    if nominalLoc == 0:
        return dist.DDist({nominalLoc : 0.9, nominalLoc + 1 :
0.1})
    elif nominalLoc == hallwayLength - 1:
        return dist.DDist({nominalLoc : 0.9, nominalLoc - 1 :
0.1})
    else:
        return dist.DDist({nominalLoc : 0.8, nominalLoc + 1 :
0.1, nominalLoc -1 : 0.1})

def black(actualColor):
    return dist.DDist({'black' : 1.0})
noisyTransModel = makeTransitionModel(standardDynamics,
noisyTrans)
```

The ringDynamics function calculates the circular dynamic position update based on the given action to ensure the robot's domestic circulation in the corridor. LeftSlipTrans and noisyTrans respectively define left-shift and noisy transition models, describing the probability distribution of the robot moving up or offset at the current position, where the noise transition model has different transition probabilities at different positions (such as both ends of the corridor). The black function always returns black observation results, which are

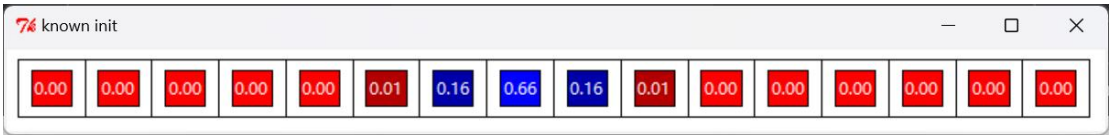
used to simulate a constant "black" sensor. Finally, noisyTransModel is a transition model created based on standardDynamics and noisyTrans functions, which comprehensively considers dynamic behavior and noisy state transition processes. Together, these functions and models constitute a dynamic environment that simulates the robot's movement and perception in a noisy corridor environment.

Step 6: Testhallway

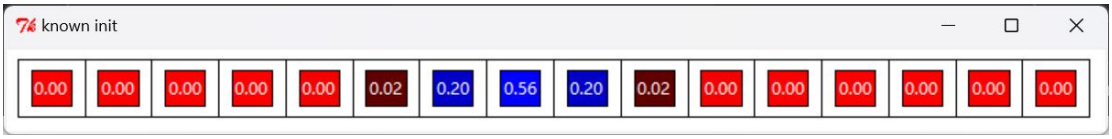
If we keep inputting 0:



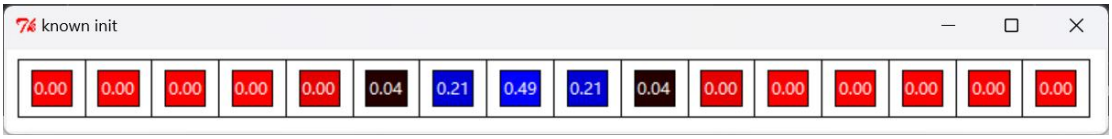
First 0



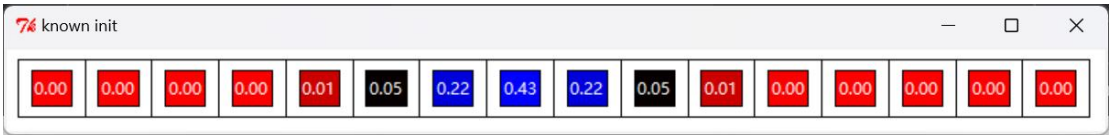
Second 0



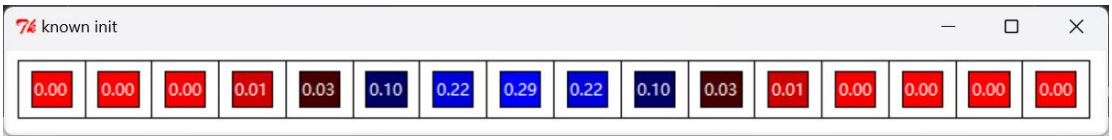
Third 0



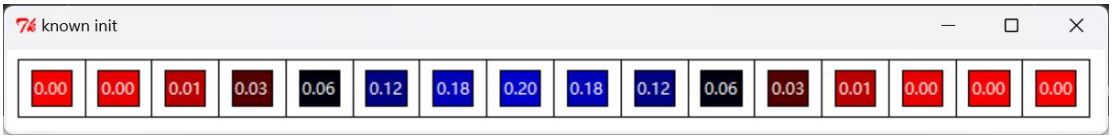
Fourth 0



Fifth 0



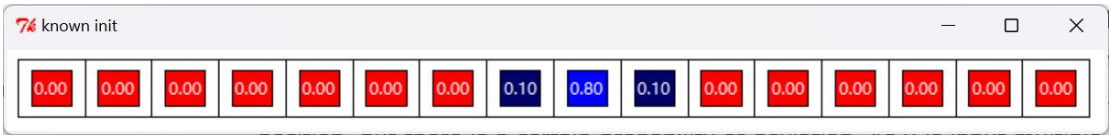
Tenth 0



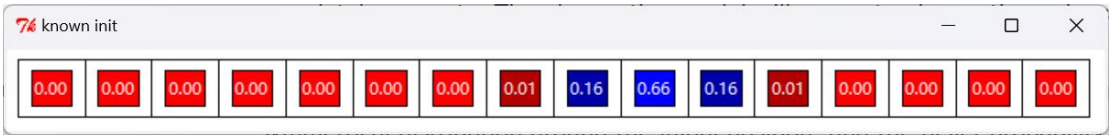
Twentieth 0

From the above results, we can know that when multiple zeros are input: although the input action is 0, due to transfer noise, the current position may shift with a small probability. Each observation may not be completely accurate. The observation model will generate observation values that are close to the actual position, but there is a certain probability of deviation. As 0 is input multiple times, the transfer noise will gradually spread out the distribution. The initial centralized distribution will gradually expand, forming a symmetrical distribution around the initial position, and the belief probability will decrease as the distance increases. Observation noise makes the distribution of observation results more blurred, and the concentration gradually decreases. Over time, the belief state tends to be a balanced distribution, and the observation results show the characteristics of a Gaussian distribution, symmetrical around position 7 and attenuating to both sides.

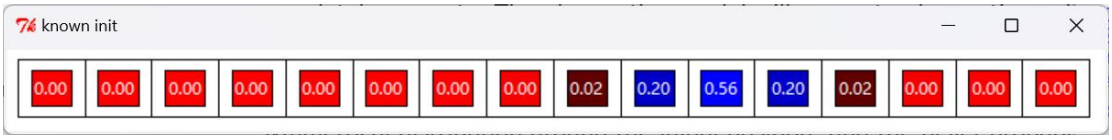
If we keep inputting 1:



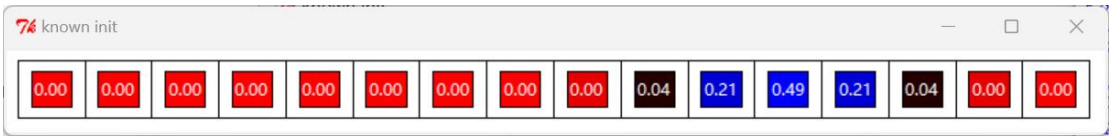
First 1



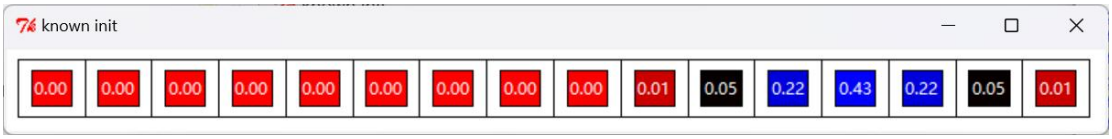
Second 1



Third 1



Fourth 1



Fifth 1

From the above results, we can know that if we drive the robot around, the belief distribution will expand along the robot's motion trajectory and be constrained by the circular corridor. If the observation noise is small, the observation results can still maintain a high correlation with the actual position, and the belief distribution tends to be stable over time. The observation noise causes a certain uncertainty when the robot updates its belief based on the observation, making it difficult to accurately locate.

Checkoff 1.

How do whiteEqGreenObsDist and whiteVsGreenObsDist observation models (from Check Yourself 3) compare to:

• A perfect sensor model

• A sensor that always reads ’black’ no matter what room it is in

Demonstrate the world with noisy dynamics from Step 6 to a staff member and explain why it does what it does.

1. Comparison with the Perfect Sensor Model
    - The perfect sensor model provides accurate information about the actual color of the room. If the room is actually white, the probability of observing white is 1 and 0 for other colors; if it's green, the probability of observing green is 1 and 0 for others.
    - The whiteEqGreenObsDist model, when the actual color is white or green, assigns a probability of 0.5 to observing white and green respectively and 1 to other colors. This is different from the perfect sensor model as it fails to precisely distinguish between white and green rooms, introducing more uncertainty in the robot's belief about its location.
    - The whiteVsGreenObsDist model always misidentifies white rooms as green and vice versa, which is completely contrary to the perfect sensor model and will lead to significant wrong in the robot's belief about its location.
2. Comparison with the Sensor that Always Reads 'Black'
    - The sensor that always reads 'black' provides no useful information about the actual color of the room, regardless of the room's true color.
    - The whiteEqGreenObsDist model, although it has inaccuracies in distinguishing white and green, still has a chance to observe white or green. Compared to the sensor that always reads 'black', it offers more possibilities about the room color. The robot can adjust its belief state based on the probabilities of

observing white or green, albeit inaccurately.

- The whiteVsGreenObsDist model also provides more information than the sensor that always reads 'black'. It gives wrong but specific observations (observing green in a white room and vice versa), allowing the robot to attempt to infer its location based on this incorrect but patterned observation, although the initial judgment is wrong and requires more information to correct.

3. Explanation of the World with Noisy Dynamics in Step 6

   - In Step 6, a test world with only white rooms and noisy transitions and observations is created, and the state estimator is initialized with a probability of 1 assigned to location 7. When the action 0 (attempting to stay in the current position) is selected multiple times, due to the possible noise in the observation model (such as whiteEqGreenObsDist or whiteVsGreenObsDist), the robot's observations may not be accurate. Even if the actual position remains unchanged, the observed color may not match the actual situation, leading to incorrect updates in the belief state. For example, if using the whiteVsGreenObsDist model, when the robot is actually in a white room (location 7), it may observe green, causing it to wrongly believe that it might be in a green room and adjust its belief state, reducing the probability estimate for location 7 and increasing it for other locations (which might be considered as possible green room locations). When the robot is driven to move, the noisy dynamics affect its actual movement. It may not move to the expected position accurately as commanded. For instance, when the action is to move one room to the right, due to noise, it may move to other positions or stay in the same place, further exacerbating the mismatch between the belief state and the actual position. The robot's belief state continuously updates based on inaccurate observations and unreliable movements, making its estimate of its location more uncertain and inaccurate. The probability distribution across the hallway becomes more dispersed and chaotic until sufficient information (observations and actions) is obtained to gradually correct the belief state to be closer to the actual position.

Step 7: Do the problem

The answer is as follows:

Part 1:

1. What is the robot's prior belief $B_0(s) = Pr(S_0 = s)$ for each of the states *s*?
   $B_0(s) = Pr(S_0 = s)$
1/3	1/3	1/3

2. First, the robot makes an observation. Let's assume it sees 'white', because it is in a white room and there's no sensor noise. So, $O_0 = white$). We want to know what is the new belief state after this observation?
   - First, figure out $Pr(O_0 = white|S_0 = s)$ for each state *s*
1	0	1
   - Then compute $Pr(O_0 = white|S_0 = s)Pr(S_0 = s)$ for each state *s*. Note that this is the same as $Pr(O_0 = white, S_0 = s)$
1/3	0	1/3
   - Now, compute $Pr(O_0 = white)$.
2/3
   - Compute the new belief state after the observation; using the definition of conditional probability and the previous two results:
     $B'_0(s) = Pr(S_0 = s|O_0 = white)$.
1/2	0	1/2

3. If we told the robot to go right one room with action $I_0 = 1$, what would the belief state be after taking the state transition into account?
   $B_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1)$
0	1/2	1/2

4. Now, assume the robot observes 'green' because it's in a green room and there's no noise, $O_1 = green$). What will the belief state be after this?
   $B'_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1, O_1 = green)$
0	1	0

5. If we told the robot to go right with action $I_1 = 1$, what would the belief state be after taking the state transition into account?
   $B_2(s) = Pr(S_2 = s|O_0 = white, I_0 = 1, O_1 = green, I_1 = 1)$
0	0	1

Part 2:

1. What is the robot's prior belief for each of the states *s*?
   $B_0(s) = Pr(S_0 = s)$
   1/3 | 1/3 | 1/3

2. What is the distribution over what the robot sees? That is, what is $Pr(O_0 = o)$ for all possible colors *o*?
   $P(O_0 = black) =$ 1/20 $P(O_0 = white) =$ 11/20 $P(O_0 = red) =$ 1/20
   $P(O_0 = green) =$ 3/10 $P(O_0 = blue) =$ 1/20

3. First, the robot makes an observation. Let's assume it sees 'white'. So, $O_0 = white)$. We want to know the new belief state after the observation.
   $B'_0(s) = Pr(S_0 = s|O_0 = white)$.
   16/33 | 1/33 | 16/33

4. If we told the robot to go right $I_0 = 1$, what would the belief state be after taking the state transition into account? Recall that motion is perfect.
   $B_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1)$
   0 | 16/33 | 17/33

5. Now, what is the distribution over what the robot sees? That is, what is $Pr(O_1 = o)$ for all possible colors *o*?
   $P(O_1 = black) =$ 1/20 $P(O_1 = white) =$ 24/55 $P(O_1 = red) =$ 1/20
   $P(O_1 = green) =$ 91/220 $P(O_1 = blue) =$ 1/20

6. Now, assume the robot sees 'white' again, $O_1 = white)$. What will the belief state be after this?
   $B'_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1, O_1 = white)$
   0 | 1/18 | 17/18

7. If we told the robot to go right $I_1 = 1$, what would the belief state be after taking the state transition into account?
   $Pr(S_2 = s|O_0 = white, I_0 = 1, O_1 = white, I_1 = 1)$
   0 | 0 | 1

8. If instead of having seen 'white' and gone right (as above), the robot had seen 'green' and gone right, what would the belief state be? That is, what is
   $Pr(S_2 = s|O_0 = white, I_0 = 1, O_1 = green, I_1 = 1)$
   0 | 0 | 1

Step 8: Do the problem

The answer is as follows:

1. What is the robot's prior belief $B_0(s) = Pr(S_0 = s)$ for each of the states s?
   $B_0(s) = Pr(S_0 = s)$
   1/3 1/3 1/3
2. First, the robot makes an observation. Let's assume it sees 'white'. So, $O_0 = white$. We want to know the new belief state after the observation
   $B'_0(s) = Pr(S_0 = s|O_0 = white)$.
   16/33 1/33 16/33
3. If we told the robot to go right $I_0 = 1$, what would the belief state be after taking the state transition into account?
   $B_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1)$
   8/165 29/66 169/330
4. Now, assume the robot sees 'white' again, $O_1 = white$. What will the belief state be after this?
   $B'_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1, O_1 = white)$
   256/3105 29/621 2704/3105
5. If we told the robot to go right $I_1 = 1$, what would the belief state be after taking the state transition into account?
   $B_2(s) = Pr(S_2 = s|O_0 = white, I_0 = 1, O_1 = white, I_1 = 1)$
   128/15525 4897/31050 25897/31050

Step 9: Wk.11.1.6: Sonar hit

Here is a basic idea of how we implemented the code (not the actual code):

```
Python
def sonar_hit(distance, sonar_pose, robot_pose):
    # Obtain angles measured by sonar and robot posture
    thetaS = sonar_pose[2]
    xR, yR, thetaR = robot_pose

    # Calculate the position of the sonar hit point in the
robot coordinate system
    bx = distance * math.cos(thetaS)
    by = distance * math.sin(thetaS)

    # Convert the point from the robot coordinate system to the
global coordinate system
    ax = xR + math.cos(thetaR) * bx - math.sin(thetaR) * by
    ay = yR + math.sin(thetaR) * bx + math.cos(thetaR) * by

    return (ax, ay)
```

Based on the above ideas, we can obtain the following code:

```
Python
def sonarHit(distance, sonarPose, robotPose):
```

```
return(robotPose.transformPose(sonarPose.transformPoint(util.Point(distance, 0))))
```

`util.Point(distance, 0)`: This creates a point that represents the position of the object measured by sonar in the sonar coordinate system. The x-coordinate of this point is distance, and the y-coordinate is 0, which means the straight-line distance of the object in front of the sonar.

`sonarPose.transformPoint(...)`: This method converts the input point (distance, 0) from the sonar coordinate system to the robot coordinate system. In fact, assuming a certain position and orientation (sonarPose) of the sonar in the robot coordinate system, the point is converted from the local coordinate system of the sonar to the robot coordinate system through transformPoint.

`robotPose.transformPose(...)`: This method further converts the converted point (position in the robot coordinate system) from the local coordinate system of the robot to the global coordinate system. The position and orientation of the robot in the global coordinate system are provided by robotPose, and transformPose converts the point to the global coordinate system based on the current position and orientation of the robot.

The function of this function is to calculate the position of the object reflected by the sonar beam in the global coordinate system based on the distance measured by the sonar, the position and orientation of the sonar, and the position and orientation of the robot.

The implementation of the code is completed through two consecutive coordinate transformations. First, the sonar measurement value is converted from the sonar coordinate system to the robot coordinate system, and then it is converted from the robot coordinate system to the global coordinate system.

In this way, the sonarHit function can return the position of the object corresponding to the sonar measurement value in the global coordinate system.

Step 10: Wk.11.1.7: Ideal Sonar Readings

If sonarReading is greater than sonarMax, return the maximum index (numObservations - 1), because readings exceeding sonarMax should be classified into the last interval.

Otherwise, we need to map the readings to the range of [0, sonarMax] and divide them into numObservations intervals according to numObservations. The width of each interval is sonarMax/numObservations.

Use the formula to calculate which interval the reading belongs to and return the index of that interval.

```
Python
import lib601.util as util

def sonarHit(distance, sonarPose, robotPose):
    return robotPose.transformPoint(sonarPose.transformPoint(\
```

```
util.Point(distance,0)))

def idealReadings(wallSegs, robotPoses):
    readings = []
    for pose in robotPoses:
        sensorOriginPoint = sonarHit(0, sonarPose, pose)
        sonarRay = util.LineSeg(sensorOriginPoint,
sonarHit(sonarMax, sonarPose, pose))
        hits = [(seg.intersection(sonarRay), seg) for seg in
wallSegs]
        distances = [sensorOriginPoint.distance(hit) for (hit,
seg) in hits if hit]
        if distances:
            idealReading = min(distances)
        else:
            idealReading = sonarMax
        readings.append(discreteSonar(idealReading))
    return readings

def discreteSonar(sonarReading):
    return min(numObservations - 1, int(sonarReading / sonarMax
* numObservations))

sonarMax = 1.5
numObservations = 10
sonarPose = util.Pose(0.08, 0.134, 1.570796)
# sonarPose = util.Pose(0,0,3.14/2)
def wall(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return util.LineSeg(util.Point(x1,y1), util.Point(x2,y2))
wallSegs = [wall((0, 2), (8, 2)),
            wall((1, 1.25), (1.5, 1.25)),
            wall((2, 1.75), (2.8, 1.75))]
robotPoses = [util.Pose(0.5, 0.5, 0),
              util.Pose(1.25, 0.5, 0),
              util.Pose(1.75, 1.0, 0),
              util.Pose(2.5, 1.0, 0)]

print("discreteSonar")
print(discreteSonar(0.1))
print(discreteSonar(0.33))
print(discreteSonar(0.55))

print("idealReading")
print(idealReadings(wallSegs, robotPoses))
```

And the output of this processor is as blows:

> discreteSonar
>
> 0
>
> 2
>
> 3
>
> idealReading
>
> [9, 4, 5, 4]

Step 11: Wk. 12.2.3 Localization

Here is the answer to Wk. 12.2.3 Localization in swLab12.

1. `Preprocessor (at time 0):`
   - Input: an instance of `io.SensorInput`:
     - `sonars = (0.8, 1.0, ...)`
     - `odometry = Pose(1.0, 0.5, 0.0)`
   - Output: a tuple `(obs, act)`; if the output is `None`, enter `None` in both boxes.
     - `obs =` 5
     - `act =` None
2. `Preprocessor (at time 1):`
   - Input: an instance of `io.SensorInput`:
     - `sonars = (0.25, 1.2, ...)`
     - `odometry = Pose(2.4, 0.5, 0.0)`
   - Output: a tuple `(obs, act)`; if the output is `None`, enter `None` in both boxes.
     - `obs =` |
     - `act =` |
3. `Preprocessor (at time 2):`
   - Input: an instance of `io.SensorInput`:
     - `sonars = (0.16, 0.2, ...)`
     - `odometry = Pose(7.3, 0.5, 0.0)`
   - Output: a tuple `(obs, act)`; if the output is `None`, enter `None` in both boxes.
     - `obs =` |
     - `act =` 5
4. `Estimator (at time 0):`
   - Input: `(obs, act)` the output tuple from Preprocessor
   - Output: probability distribution over robot states (x indices)

1/10	1/10	1/10	1/10	1/10	1/10
1/10	1/10	1/10	1/10		

5. `Estimator (at time 1):`
    - Input: `(obs, act)` the output tuple from Preprocessor
    - Output: probability distribution over robot states (x indices)

0	0	1/6	1/6	0	1/6
1/6	1/6	0	1/6		

6. `Estimator (at time 2):`
    - Input: `(obs, act)` the output tuple from Preprocessor
    - Output: probability distribution over robot states (x indices)

0	0	0	0	0	0
0	1/3	0	2/3		

3. Summary

- In this experiment, we are dedicated to researching the state estimation of robots in corridor environments. The core of this dl is to accurately infer the probability distribution of the robot's hidden state based on input actions and observation data, and then determine its pose using the knowledge of hidden Markov chains
- We deeply study and apply the key knowledge of probability state estimation, including initial distribution, observation model, and transition model, which are the theoretical pillars of the entire experiment and design diverse models such as whiteEqGreenObsList, whiteVsGreenObsList, and noisyOs to simulate the probability patterns of observation data under various complex error scenarios.
- For the transfer model, We designed models such as ringDynamics, leftLipsTrans, and noisyTrans to accurately characterize the probability characteristics of robot position transfer under different action instructions.
- By learning and relying on the principle of coordinate transformation, we successfully implemented the sonar Hit function to accurately calculate the coordinate information of the ultrasound reflecting object, adding a key dimension to the robot's perception of the surrounding environment;
- Based on the reading processing logic, I effectively used the idealRatings function to convert ultrasound readings into discrete interval indexes, significantly improving data processing efficiency and positioning accuracy;
- Finally, we efficiently processed sensor input data, successfully obtained the probability distribution of robot states, and achieved accurate and reliable positioning targets.